NPS52-86-004

NAVAL POSTGRADUATE SCHOOL

Monterey, California

EXPERIENCE WITH Ω

IMPLEMENTATION OF A
PROTOTYPE PROGRAMMING ENVIRONMENT
PART III

Bruce J. MacLennan

January 1986

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. H. Shumaker
Superintendent

D. A. Schrady
Provost

The work reported herein was supported by Contract from the Office of Naval Research.

This report was prepared by:

VINCENT Y. LUM
Chairman
Department of Computer Science

KNEALE T. MARSHALL
Dean of Information and
Policy Science

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-86-004	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: EXPERIENCE WITH Ω IMPLEMENTATION OF A PROTOTYPE PROGRAMMING ENVIRONMENT PART III	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Bruce J. MacLennan	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943-5100	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61153N; RR014-08-01, N0001485WR24057
11. CONTROLLING OFFICE NAME AND ADDRESS: Chief of Naval Research, Arlington, VA 22217	12. REPORT DATE: January 1986
	13. NUMBER OF PAGES: 46
14. MONITORING AGENCY NAME & ADDRESS *(If different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This is the third report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. The present report extends the interpreter, unparser, syntax directed editor, command interpreter and debugger to accommodate comments (which are used both statically and dynamically) and conditional expressions. A running implementation of these ideas is listed in the appendices.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

EXPERIENCE WITH Ω

IMPLEMENTATION OF A

PROTOTYPE PROGRAMMING ENVIRONMENT

PART III

Bruce J. MacLennan
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract:

This is the third report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. The present report extends the interpreter, unparser, syntax directed editor, command interpreter and debugger to accommodate comments (which are used both statically and dynamically) and conditional expressions. A running implementation of these ideas is listed in the appendices.

1. Introduction

Our goal in this series of reports* MacLennan85b, MacLennan85c is to explore in the context of a very simple language the use of the Ω programming notation MacLennan83, MacLennan85a to implement some of the tools that constitute a programming environment.

The structure of this report is as follows: First we discuss the accommodation of comments in the abstract structure of the program. This demonstrates how documentation and nonprogram text can be incorporated into the program structure. These comments are used both statically (i.e., as commentary in the source code) and dynamically (i.e., to provide informative information at runtime). Second we introduce a conditional expression into the language. This involves no new ideas, but is essential if the recursive functions introduced in Part IV are to be useful.

* Support for this research was provided by the Office of Naval Research under contract N00014-86-WR-24092.

2. Comments

2.1 Requirements

How can we include comments in our programs? In particular, how can comments be incorporated into the abstract program structure? The solution to this problem will indicate the solution to a more general problem, namely, how documentation, specifications, explanations, etc. can be related to the program structure.

Our solution here is to define a new relation 'Comment' that is used to link a comment to any program node. Its properties are:

- Comment (S, E)

  S is the comment on E

- Degree (Comment, 2)

- Function (Comment, expr, string)

The following transcript is an example of creating and displaying comments:

```
show
 [let X = (3+5)
   let Y = (6÷2)
    (X+(Y-1))]

rem "Distance"
show
 [let {Distance}
   X = (3+5)
   let Y = (6÷2)
    (X+(Y-1))]
```

Notice that the comment "Distance" is attached to the current node (a **let** in this case), and that this alters the formatting of that node. The presumption is that the comment on a **let** documents the name defined by that **let**. Next we move to the inner **let** and attach a comment to it:

```
in
  [let Y = (6÷2)
     (X+(Y−1)) ]

rem "Altitude"
```

Comments can also be attached to operator application nodes, as the following illustrate:

```
in
  (X+(Y−1))
in
  X
next
  (Y−1)
rem "offset"
root
  let {Distance}
   X = (3+5)
    [let {Altitude}
      Y = (6÷2)
        (X+ {offset}(Y−1) ) ] ]
```

2.2 Editor Commands

The **rem** command, which attaches a comment to a program node, is straight-forward to implement:

*Command(**rem**), *Argument(S), CurrentNode(E), ¬Comment(−,E)

⇒ Comment(S,E).

*Command(**rem**), *Argument(−), CurrentNode(E), Comment(−,E)

⇒ Display("node already commented").

Note that we allow only one comment to be attached to a node. This is a debatable restriction, but it seems that permitting multiple comments would require some strategy for organizing them. This is left

as an exercise for the reader.

We also need a **delete_rem** command for uncommenting a node:

*Command(**delete_rem**), CurrentNode(E), *Comment($-$, E)

$\Rightarrow$ Display("done").

*Command(**delete_rem**), CurrentNode(E), ¬Comment($-$, E)

$\Rightarrow$ Display("no comment").

2.3 Unparsing Comments

Unfortunately, with our present organization, most of the unparsing rules must be modified to incorporate comments into the unparsed program representation. Is there a way of unparsing comments without duplicating all the unparsing rules? This is left as an exercise for the reader.

First we consider the rule for unparsing comments on operator applications. Our goal is that an addition node with a comment be displayed like this:

$$\{S\}\,(X + Y)$$

This is easily accomplished by the rule:

Appl(E), Op(N,E), Left(X,E), Right(Y,E), *Image(U,X), *Image(V,Y), Comment(S,E)

$\Rightarrow$ Image("{" ^ S ^ "}(" ^ U ^ N ^ V ^ ")", E)

else previous Appl rule.

The only difference from the uncommented case is the incorporation of the comment into the image. Unparsing comments on Vars and Cons is done in essentially the same way as those on Appls.

Next we consider unparsing comments on blocks. The goal is a display of the form:

let $\{S\}$

$N = X$

B

Again, all we have to do is provide an unparsing rule like the uncommented one, except that the

comment is incorporated in the image:

Block(E), BndVar(N,E), BndVal(X,E), Body(B,E),

*Image(U,X), *Image(V,B), Comment(S,E)

$\Rightarrow$ Image(

TabIn ^NL ^"{**let** {" ^S ^"}"

^TabIn ^NL ^N ^" = "U

^NL ^V ^" }"

^TabOut ^TabOut, E)

else previous rule.

2.4 Dynamic Use of Comments

Since a comment on a block presumably describes the purpose of the local variable, it would be convenient to include this comment in the dynamic context, so that it can be displayed by the debugger. This is accomplished by modifying the evaluator so that the comment on a block is incorporated in the context (by being attached as a 'Comment' to the context). The rule that does this is:

Block(E), BndVar(N,E), BndVal(X,E), Body(B,E), *Value(V,X,C), *Avail(D), Comment(S,E)

$\Rightarrow$ Context(D), Binds(D,N,V), Nonlocals(C,D), Eval(B,D), Comment(S,D)

else previous rule.

There is no point in making comments part of the dynamic structure unless we are going to make use of them. For example, we can modify the debugger so that it displays any comments attached to a binding, so that we will know the purpose of the variable. Here is an example transcript that makes use of the example program shown above:

context

Y = 3 {Altitude}

out_context

X = 8 {Distance}

This modification of the debugger is accomplished by adding the following rule:

*Command(**context**), CurrentContext(C), Binds(C,N,V), Comment(S,C)

$\Rightarrow$ Display(N ^" = " ^string←int[V] ^" {" ^S ^"}", E)

else previous rule.

The reader should try to think of other possible runtime applications of comments and other nonprogram information.

3. Conditional Expressions

3.1 Concrete and Abstract Syntax

Next we introduce a conditional expression into our language. This construct will be necessary for effective use of the recursive functions introduced in Part IV. A conditional expression is displayed in the following format:

(**if** B
 then T
 else F)

In the context of a larger program it would look like this:

[**let** N = (X−1)
 (**if** (N > 0)
  then 1
  else (N * (N − 1)))

Note that we will also have to extend our operator application syntax to include relational operators ('=', '>', etc.); this is left as an exercise for the reader.

The abstract structure of conditional expressions is represented by the following relations:

- ConEx(E)

  E is a conditional expression

Degree(ConEx,1).

- Cond(B,E)

  B is the condition of E

  Function(Cond, ConEx, expr).

- Conseq(T,E)

  T is the consequent of E

  Function(Conseq, ConEx, expr).

- Alt(F,E)

  F is the alternate of E

  Function(Alt, ConEx, expr).

3.2 Editor Commands

It's necessary to incorporate a new command into the editor so that we can create ConEx nodes. The rule for the '**if**' command is routine:

*Command(**if**), *CurrentNode(E), *Undef(E), *Avail(B,T,F)

$\Rightarrow$ ConEx(E), Cond(B,E), Conseq(T,E), Alt(F,E),
Undef(B), Undef(T), Undef(F), CurrentNode(B).

The regularity of all these node creation commands (**if**, **let**, **var**, etc.) suggests that there ought to be a regular way of handling them. This too is left as an exercise for the reader.

The editor must also be augmented with rules for the **in**, **out**, **next** and **prev** commands on ConEx nodes. The reader should also consider how this proliferation of rules can be avoided. A typical rule is the **in** rule:

*Command(**in**), *CurrentNode(E), ConEx(E), Cond(B,E)

$\Rightarrow$ CurrentNode(B), Command(**show**).

3.3 Unparsing

As usual we have two rules: analysis and synthesis. The analysis rule is:

*Unparse(E), ConEx(E), Cond(B,E), Conseq(T,E), Alt(F,E)

$\Rightarrow$ Unparse(B), Unparse(T), Unparse(F).

The synthesis rule gathers the images of the subexpressions and assembles them into an image for the conditional:

ConEx(E), Cond(B,E), Conseq(T,E), Alt(F,E), *Image(U,B), *Image(V,T), *Image(W,F)

$\Rightarrow$ Image(TabIn ^NL ^

"(**if**" ^ U ^NL ^

" **then**" ^ V ^NL ^

" **else**" ^ W ^")" ^

TabOut ^NL, E).

A rule for commented ConEx nodes is left as an exercise for the reader.

3.4 Evaluation

Evaluation of the conditional node requires two separate analysis/synthesis steps. The first analysis rule requests evaluation of the condition:

*Eval(E,C), ConEx(E), Cond(B,E)

$\Rightarrow$ Eval(B,C).

When a value is returned for the condition, either the consequent or the alternate must be evaluated (depending on whether or not the condition was **true**). Thus the synthesis pass for the condition is combined with the analysis pass for either the consequent or alternate. Evaluation of the consequent when the condition is true is handled by this rule:

ConEx(E), Cond(B,E), Conseq(T,E), *Value(**true**,B,C)

$\Rightarrow$ Eval(T,C).

Evaluation of the alternate is analogous.

The final synthesis pass occurs when a value arrives at either the consequent or alternate; this value is attached to the conditional itself. The case where the value arrives at the consequent is handled by:

ConEx(E), Conseq(T,E), *Value(V,T,C)

$\Rightarrow$ Value(V,E,C).

The alternate case is analogous.

4. References

[MacLennan83] MacLennan, B. J., A View of Object-Oriented Programming, Naval Postgraduate School Computer Science Department Technical Report NPS52-83-001, February 1983.

[MacLennan84] MacLennan, B. J., The Four Forms of Ω : Alternate Syntactic Forms for an Object-Oriented Language, Naval Postgraduate School Computer Science Department Technical Report NPS52-84-026, December 1984.

[MacLennan85a] MacLennan, B. J., A Simple Software Environment Based on Objects and Relations, *Proc. of ACM SIGPLAN 85 Conf. on Language Issues in Prog. Environments*, June 25-28, 1985, and Naval Postgraduate School Computer Science Department Technical Report NPS52-85-005, April 1985.

[MacLennan85b] MacLennan, B. J., Experience with Ω : Implementation of a Prototype Programming Environment Part I, Naval Postgraduate School Computer Science Department Technical Report NPS52-85-006, May 1985.

[MacLennan85c] MacLennan, B. J., Experience with Ω : Implementation of a Prototype Programming Environment Part II, Naval Postgraduate School Computer Science Department Technical Report NPS52-85-015, December 1985.

[McArthur84] McArthur, Heinz M., *Design and Implementation of an Object-Oriented, Production-Rule Interpreter,* MS Thesis, Naval Postgraduate School Computer Science Department, December 1984.

[Ufford85] Ufford, Robert P., *The Design and Analysis of a Stylized Natural Grammar for an Object Oriented Language (Omega),* MS Thesis, Naval Postgraduate School Computer Science Department, June 1985.

APPENDIX A: Prototype Programming Environment

The following is a loadable input file for the prototype programming environment described in this report. It is accepted by the McArthur interpreter [McArthur84], which differs in a few details from the Ω notation used in this report (see [MacLennan84]). Also note that the declarations have been reordered somewhat from that in previous reports in this series; the new order better reflects the object-oriented design. A transcript of a test execution of this environment is listed in Appendix B.

```
!_____________________________________________________

!              PI-3

!

! A simple programming environment for an arithmetic

! expression language, including interpreter, unparser,

! syntax directed editor and debugger.

!

! Features included in the language:

!   - Constants

!   - Arithmetic Operations

!   - Statically Nested Declarations

!   - Comments

!   - Conditional Expressions

!_____________________________________________________

!      PERVASIVE RELATIONS

! Evaluation

newrelation {"Eval"};

newrelation {"Check"};

newrelation {"Value"};

newrelation {"Meaning"};
```

```
newrelation {"Explanation"};

! Contexts and Bindings

newrelation {"Context"};
newrelation {"Binds"};
newrelation {"Nonlocal"};
newrelation {"Looking"}.

! Unparsing

newrelation {"Unparse"};
newrelation {"Image"};
newrelation {"Template"};

! Comments

newrelation {"Comment"}.

! Format Control Constants

define {root, "NL", "
"};
define {root, "TabIn", ""};
define {root, "TabOut", ""};

! Logical Constants

define {root, "true", 1};
define {root, "false", 0}.
```

```
! COMMAND INTERPRETER

! Command Interpreter Relations

newrelation {"Command"};

newrelation {"Argument"};

newrelation {"Root"};

newrelation {"Undef"};

newrelation {"CurrentNode"};

newrelation {"CurrentContext"};

newrelation {"SuspendedEval"};

newrelation {"Break"};

newrelation {"EvalPending"};

newrelation {"ShowPending"};

newrelation {"CommandPending"};

newrelation {"CreateRoot"};

newrelation {"CreateContext"}.

define {root, "ComIntRules", << 

! evaluate Command

if *Command("evaluate"), CurrentNode(E), CurrentContext(C)

-> Eval(E,C), EvalPending(E), CommandPending(E);

if *Value(V,E,C), *EvalPending(E), *CommandPending(_)

-> displayn {V};

! Error Handler

if *Break(M,E,C), *CommandPending(_), *EvalPending(R), *SuspendedEval(_)
```

```
-> displayn{M}, SuspendedEval(R), CurrentNode(E), CurrentContext(C);

! resume Command

if *Command("resume"), SuspendedEval(Nil)

-> displayn{"no evaluation in progress"}

else if *Command("resume"), CurrentNode(E), CurrentContext(C), *SuspendedEval(R)

-> Eval(E,C), EvalPending(R), SuspendedEval(Nil);

! return Command

if *Command("val"), *Argument(V), CurrentNode(E)

-> Value(V,E,C);

! show Command

if *Command("show"), CurrentNode(E)

-> Unparse(E), ShowPending(E), CommandPending(E);

if *Image(S,E), *ShowPending(E), *CommandPending(-)

-> displayn{S};

! abort Command

if Command("abort"), *Eval(E,C) -> ;

if Command("abort"), *Value(V,E,C) -> ;

if Command("abort"), *Check(V,E,C) -> ;

if Command("abort"), *Nonlocal(C,D) -> ;

if Command("abort"), *Binds(D,N,V) -> ;

if *Command("abort"),
```

```
~Eval(E,C), ~Value(V,E,C), ~Nonlocal(C,D), ~Binds(D,N,V),
*SuspendedEval(- ), *CurrentContext(- )
-> CurrentContext(Nil), SuspendedEval(Nil), displayn{"aborted"};

! done Command

if *Command("done") -> displayn{"PI system stopped"};
```

```
! Syntax Directed Editing

if *Command("delete"), CurrentNode(E), Undef(E)
-> displayn("already deleted");

! begin Command

if *Command("begin"), *CurrentNode(Q)
->  CreateRoot(newobj{}), CommandPending(Nil);

if *CreateRoot(E), *CommandPending(-)
-> Root(E), Undef(E), CurrentNode(E);

! root Command

if *Command("root"), *CurrentNode(Q), Root(E)
-> CurrentNode(E), Command("show");

! Debugging Commands

! context Command

if *Command("context"), CurrentContext(C), Binds(C,N,V), Comment(S,C)
-> displayn( N + " = " + int_str(V) + " {" + S + "}" )

else if *Command("context"), CurrentContext(C), Binds(C,N,V)
-> displayn( N + " = " + int_str(V) )

else if *Command("context")
-> displayn("no bindings");

! out_context Command

if *Command("out_context"), *CurrentContext(D), Nonlocal(C,D)
```

```
-> CurrentContext(C), Command("context")

else if *Command("out_context")

-> displayn ("at outermost level");

! in_context Command

if *Command("in_context"), *CurrentContext(C), Nonlocal(C,D)

-> CurrentContext(D), Command("context")

else if *Command("in_context")

-> displayn("at innermost level");

! alter Command

if *Command("alter"), *Argument(U),

 CurrentContext(C), *Binds(C,N,V)

-> Binds(C,N,U), Command("context")

else if  *Command("alter"), *Argument(Q)

-> displayn("no binding");

>> }.

act {ComIntRules}.
```

```
! COMMENTS

define {root, "RemRules", < <

! rem Command

if *Command("rem"), *Argument(S), CurrentNode(E), ~Comment(- ,E)
-> Comment(S,E);

if *Command("rem"), *Argument(- ), CurrentNode(E), Comment(- ,E)
-> displayn( "node already commented");

! delete_rem Command

if *Command("delete_rem"), CurrentNode(E), *Comment(- ,E)
-> displayn("done");

if *Command("delete_rem"), CurrentNode(E), ~Comment(- ,E)
-> displayn("no comment");
> > }.

act {RemRules}.


! INCOMPLETE PROGRAM

! Tables

Explanation ("Incomplete program", ["error", 0]).

define {root, "IncomProgRules", < <

! Evaluation

if *Eval(E,C), Undef(E), *CurrentNode(Q)
-> Break("Incomplete", E, C);
```

```
! Unparsing

if *Unparse(E), Undef(E)

-> Image ("< expr> ", E);

> > }.

act {IncomProgRules}.
```

```
! CONSTANT NODES

! Relations

newrelation {"Con"};

newrelation {"Litval"}.

! Functions

fn Id [x]: x.

! Tables

Meaning (Id, "lit").

Template (int_str, "lit").

define {root, "ConRules", < <

! Evaluation

if *Eval(e,c), Con(e), Litval(v,e), Meaning(f, "lit")

-> Value(f[v], e, c);

! Unparsing

if *Unparse(e), Con(e), Litval(v,e), Template(f, "lit"), Comment(s,e)

-> Image( f[v] + " {" + s + "}", e)

else if *Unparse(e), Con(e), Litval(v,e), Template(f, "lit")

-> Image(f v , e);

! # Command

if *Command("#"), *Argument(V), IsInt V , CurrentNode(E), *Undef(E)

-> Con(E), Litval(V,E);
```

```
if *Command("#"), *Argument(V), CurrentNode(E), ~Undef(E)
-> displayn("defined node");

! delete Command

if *Command("delete"), CurrentNode(E), *Con(E), *Litval(V,E)
-> Undef(E), Command("show");
>> }.

act {ConRules}.
```

```
! VARIABLE NODES

! Relations

newrelation {"Var"};

newrelation {"Ident"}.

define {root, "VarRules", < <

! Evaluation

if *Eval(E.C), Var(E), Ident(N,E)

-> Looking(N,C,E,C):

if *Looking(N,C,E,D), Binds(C,N.V)

-> Value(V,E,D)

else if *Looking(N,C,E.D), Nonlocal(Cprime,C)

-> Looking(N,Cprime,E,D)

else if *Looking(N,C,E.D), *CurrentNode(Q), *CurrentContext(Q)

-> Break("Unbound: " + N, E, D);

! Unparsing

if *Unparse(E), Var(E), Ident(N.E), Comment(S,E)

-> Image( N + " {" + S + "}", E)

else if *Unparse(E), Var(E), Ident(N,E)

-> Image(N,E);

! var Command

if *Command("var"), *Argument(N), CurrentNode(E), *Undef(E)

-> Var(E), Ident(N,E);
```

```
! delete Command

if *Command("delete"), CurrentNode(E), *Var(E), *Ident(N,E)

-> Undef(E), Command("show");

>> }.

act {VarRules}.
```

```
! APPLICATION NODES

! Relations

newrelation {"Appl"};

newrelation {"Op"};

newrelation {"Left"};

newrelation {"Right"};

newrelation {"CreateAppl"}.

! Evaluation Functions

fn Sum [x,y]: x + y;

fn Dif [x,y]: x - y;

fn Product [x,y]: x * y;

fn Quotient [x,y]:

   if y = 0 -> ["error", 1]

   else x / y;

fn Equal [x,y]: if x = y -> true else false;

fn IsErrorcode [w]:

   if ~IsList[w] | w = Nil -> Nil

   else first[w] = "error";

! Unparsing Functions

fn upSum [x,y]: "(" + x + " + " + y + ")";

fn upDif [x,y]: "(" + x + " - " + y + ")";

fn upProd [x,y]: "(" + x + " x " + y + ")";

fn upQuot [x,y]: "(" + x + " / " + y + ")";

fn upEqua [x,y]: "(" + x + " = " + y + ")".
```

```
! Evaluation Tables

Meaning (Sum, "+ ");

Meaning (Dif, "-");

Meaning (Product, "x");

Meaning (Quotient, "/");

Meaning (Equal, "= ").

! Unparsing Tables

Template (upSum, "+ ");

Template (upDif, "-");

Template (upProd, "x");

Template (upQuot, "/");

Template (upEqua, "= ").

! Other Tables

Explanation ("division by zero", ["error", 1]).

define {root, "ApplRules", < <

! Evaluation

if *Eval(e,c), Appl(e), Left(x,e), Right(y,e)

-> Eval(x,c), Eval(y,c);

if *Value(u,x,c), *Value(v,y,c),

   Appl(e), Op(n,e), Left(x,e), Right(y,e), Meaning(f, n)

-> Check(f,u,v], e, c);

if *Check(w, e, c), ~IsErrorcode[w]

-> Value(w, e, c);
```

```
if *Check(w, e, c), IsErrorcode[w], Explanation(s, w), *CurrentNode(q)
-> Break(s, e, c);

! Unparsing

if *Unparse(e), Appl(e), Left(x,e), Right(y,e)
-> Unparse(x), Unparse(y);

! Unparsing Comments on Applications

if Appl(E), Op(N,E), Left(X,E), Right(Y,E),
 *Image(U,X), *Image(V,Y), Comment(S,E)
-> Image( " {" + S + "} (" + U + N + V + ") ", E)

else if *Image(u,x), *Image(v,y),
   Appl(e), Op(n,e), Left(x,e), Right(y,e), Template(f, n)
-> Image(f[u,v], e);


! +, -, x , /, = Commands

if *Command(op), member[op, ["+ ", "-", "x", "/", "= "]],
   *CurrentNode(E), *Undef(E)
-> CommandPending(E), CreateAppl(op, E, newobj{}, newobj{});

if *CreateAppl(op,E,X,Y), *CommandPending(E)
-> {Appl(E), Op(op,E), Left(X,E), Right(Y,E),
    Undef(X), Undef(Y), CurrentNode(X);
    Command("show")};

! delete Command

if *Command("delete"), CurrentNode(E),
  *Appl(E), *Op(N,E), *Left(X,E), Right(Y,E)
```

```
-> Undef(E), Command("show");

! in Command

if *Command("in"), *CurrentNode(E), Left(X,E)

-> CurrentNode(X), Command("show");

! out Command

if *Command("out"), *CurrentNode(X), Left(X,E)

-> CurrentNode(E), Command("show");

if *Command("out"), *CurrentNode(Y), Right(Y,E)

-> CurrentNode(E), Command("show");

! next Command

if *Command("next"), *CurrentNode(X), Left(X,E), Right(Y,E)

-> CurrentNode(Y), Command("show");

! prev Command

if *Command("prev"), *CurrentNode(Y), Right(Y,E), Left(X,E)

-> CurrentNode(X), Command("show");

>> }.

act{ApplRules}.
```

```
! BLOCK

! Relations

newrelation {"Block"};

newrelation {"BndVar"};

newrelation {"BndVal"};

newrelation {"Body"};

newrelation {"CreateLet"}.

define {root, "BlockRules", < <

! Evaluation

if *Eval(E,C), Block(E), BndVal(X,E)

-> Eval(X,C);

if Block(E), BndVar(N,E), BndVal(X,E), Body(B,E),

 *Value(V,X,C), Comment(S,E)

-> CreateContext (newobj{}, N, V, C, B, S)

else if Block(E), BndVar(N,E), BndVal(X,E), Body(B,E),

 *Value(V,X,C)

-> CreateContext (newobj{}, N, V, C, B);

if *CreateContext(D,N,V,C,B,S) -> CreateContext(D,N,V,C,B), Comment(S,D);

if *CreateContext(D,N,V,C,B)

-> Context(D), Binds(D,N,V), Nonlocal(C,D), Eval(B,D);

if Block(E), Body(B,E),

 *Value(V,B,D), *Nonlocal(C,D), *Binds(D,N,W), *Context(D)

-> Value(V,E,C);
```

```
! Unparsing

if *Unparse(E), Block(E), BndVal(X,E), Body(B,E)
-> Unparse(X), Unparse(B);

! Unparsing comments on blocks

if Block(E), BndVar(N,E), BndVal(X,E), Body(B,E),
 *Image(U,X), *Image(V,B), Comment(S,E)
-> Image(
 TabIn + NL + "|let {" + S + "}"
  + TabIn + NL + N + " = " + U
  + NL + V + "|"
  + TabOut + TabOut, E)

else if Block(E), BndVar(N,E), BndVal(X,E), Body(B,E),
 *Image(U,X), *Image(V,B)
-> Image( TabIn + NL
  + "|let " + N + " = " + U
  + TabIn + NL + V + "|"
  + TabOut + TabOut,
 E);

! let Command

if *Command("let"), *Argument(N), *CurrentNode(E), *Undef(E)
-> CommandPending(E), CreateLet(N, E, newobj{}, newobj{});

if *CreateLet (N, E, X, B), *CommandPending(E)
-> {Block(E), BndVar(N,E), BndVal(X,E), Body(B,E),
    Undef(X), Undef(B), CurrentNode(X);
    Command("show")};
```

```
! in Command

if *Command("in"), *CurrentNode(E), BndVal(X,E)
-> CurrentNode(X), Command("show");

! out Command

if *Command("out"), *CurrentNode(X), BndVal(X,E)
-> CurrentNode(E), Command("show");

if *Command("out"), *CurrentNode(B), Body(B,E)
-> CurrentNode(E), Command("show");

! next Command

if *Command("next"), *CurrentNode(X),
 BndVal(X,E), Body(B,E)
-> CurrentNode(B), Command("show");

! prev Command

if *Command("prev"), *CurrentNode(B),
 Body(B,E), BndVal(X,E)
-> CurrentNode(X), Command("show");
>> }.

act {BlockRules}.
```

```
! CONDITIONAL EXPRESSION NODES

! Relations

newrelation {"ConEx"};

newrelation {"Cond"};

newrelation {"Conseq"};

newrelation {"Alt"};

newrelation {"CreateConEx"}.

define {root, "ConExRules", < <

! Evaluation

if *Eval(E,C), ConEx(E), Cond(B,E)

-> Eval(B,C);

if ConEx(E), Cond(B,E), Conseq(T,E), *Value(true,B,C)

-> Eval(T,C);

if ConEx(E), Cond(B,E), Alt(F,E), *Value(false,B,C)

-> Eval(F,C);

if ConEx(E), Conseq(T,E), *Value(V,T,C)

-> Value(V,E,C);

if ConEx(E), Alt(F,E), *Value(V,F,C)

-> Value(V,E,C);

! Unparsing

if *Unparse(E), ConEx(E), Cond(B,E), Conseq(T,E), Alt(F,E)

-> Unparse(B), Unparse(T), Unparse(F);
```

```
if ConEx(E), Cond(B,E), Conseq(T,E), Alt(F,E),
 *Image(U,B), *Image(V,T), *Image(W,F)
-> Image( TabIn + NL +
 "(If " + U + NL +
 "then " + V + NL +
 "else " + W + ")" +
 TabOut + NL, E);

! Editing

! if Command

if *Command("if"), *CurrentNode(E), *Undef(E)
->  CommandPending(E), CreateConEx(E, newobj{}, newobj{}, newobj{});

if *CreateConEx(E,B,T,F), *CommandPending(E)
-> {ConEx(E), Cond(B,E), Conseq(T,E), Alt(F,E),
   Undef(B), Undef(T), Undef(F), CurrentNode(B);
   Command("show")};

! in Command

if *Command("in"), *CurrentNode(E), ConEx(E), Cond(B,E)
-> CurrentNode(B), Command("show");

! out Command

if *Command("out"), *CurrentNode(B), Cond(B,E), ConEx(E)
-> CurrentNode(E), Command("show");

if *Command("out"), *CurrentNode(T), Conseq(T,E), ConEx(E)
-> CurrentNode(E), Command("show");
```

```
if *Command("out"), *CurrentNode(F), Alt(F,E), ConEx(E)
-> CurrentNode(E), Command("show");

! next Command

if *Command("next"), *CurrentNode(B), Cond(B,E), Conseq(T,E)
-> CurrentNode(T), Command("show");

if *Command("next"), *CurrentNode(T), Conseq(T,E), Alt(F,E)
-> CurrentNode(F), Command("show");

! prev Command

if *Command("prev"), *CurrentNode(F), Alt(F,E), Conseq(T,E)
-> CurrentNode(T), Command("show");

if *Command("prev"), *CurrentNode(T), Conseq(T,E), Cond(B,E)
-> CurrentNode(B), Command("show");
>> }.

act {ConExRules}.
```

```
! TEST DRIVER

! Relations

newrelation {"Script"};

newrelation {"Test"}.

! Monadic Command List

define {root, "MonadicCommands",
  |"#", "val", "let", "var", "alter", "rem"|}.

define {root, "TestRules", < <

! Script Sequencer

if *Script(A,Nil), ~Command(-), ~CommandPending(-)

-> A("Script completed")

else if *Script(A,L), ~Command(-), ~CommandPending(-),
  member |first|L|, MonadicCommands|

-> { display {"... "};
     display {first |rest |L||};
     displayn {"" + first |L|};
     Command(first|L|), Argument(first|rest|L||);
     Script(A, rest,rest|L||) }

else if *Script(A,L),
  ~Command(-), ~CommandPending(-)

-> { displayn {"... " + first |L|};
     Command(first|L|);
     Script(A, rest|L|) };
```

```
! Test Scripts

if *Test(A,1) -> {
  Script{[
    "begin","rem","Distance","let","X","+ ","#",3,"next","#",5,"out","next",
    "rem","Altitude","let","Y","/","#",6,"next","#",2,"out","next",
    "+ ","var","X","next","rem","offset","/","var","Y","next","#",1,
    "root","evaluate"]};
  A("Test done");
  };

if *Test(A,2) -> {
  Script{["in","next","in","next","in","next","in","next",
    "delete","rem","error!","#",0,"root","evaluate"]};
  A("Test done");
  };

if *Test(A,3) -> {
  Script{["context","out_context","alter",7,
    "in_context","out_context","out_context","abort","show"]};
  A("Test done");
  };

if *Test(A,4) -> {
  Script{["delete","if","= ","var","Y","next","#",0,"out","next",
    "#",1,"next","- ","var","X","next","#",1,"root","evaluate"]};
  A("Test done");
  }

}.
```

```
act {TestRules}.

! Initialize Data Structures

CurrentNode(Nil).

CurrentContext(Nil).

SuspendedEval(Nil).

displayn{"PI-3 System loaded"}.
```

2.

1.

APPENDIX B: Transcript of Ω Session

The following is a transcript of an Ω session illustrating the operation of the prototype programming environment shown in Appendix A. The assertion 'Script {testscript}' causes the commands in testscript to be executed in order. The *n*th testscript is executed by 'Test{*n*}'. Each command is printed on a separate line, followed by whatever output is generated by the programming environment. This transcript was produced by the McArthur interpreter [McArthur84].

```
% omega
OMEGA-1  11/30/84

Use Cntl-D or exit{} to quit.

For help, enter help{"?"}.

To report a bug, enter Bugs{}.
newrelation rule activated.
> do{"PI3.rul"}.
PI-3 System loaded
OK
> {Test{1}; Test{2}; Test{3}; Test{4}; Script{"done"}}.
 ... begin
 ... Distance rem
 ... X let
< expr>
 ... +
< expr>
 ... 3 #
 ... next
< expr>
 ... 5 #
```

... out

(3 + 5)

... next

< expr>

... Altitude rem

... Y let

< expr>

... /

< expr>

... 6 #

... next

< expr>

... 2 #

... out

(6 / 2)

... next

< expr>

... +

< expr>

... X var

... next

< expr>

... offset rem

... /

< expr>

... Y var

... next

< expr>

```
... 1 #
... root
|let {Distance}
 X = (3 + 5)
 |let {Altitude}
  Y = (6 / 2)
  (X + {offset} (Y/1) ) ] ]
... evaluate
11
... in
(3 + 5)
... next
|let {Altitude}
 Y = (6 / 2)
 (X + {offset} (Y/1) ) |
... in
(6 / 2)
... next
(X + {offset} (Y/1) )
... in
X
... next
{offset} (Y/1)
... in
Y
... next
1
... delete
```

```
< expr>
... error! rem
... 0 #
... root
[let {Distance}
 X = (3 + 5)
 [let {Altitude}
   Y = (6 / 2)
   (X + {offset} (Y/0 {error!}) ) ] ]
... evaluate
division by zero
... context
Y = 3 {Altitude}
... out_context
X = 8 {Distance}
... 7 alter
X = 7 {Distance}
... in_context
Y = 3 {Altitude}
... out_context
X = 7 {Distance}
... out_context
no bindings
... abort
aborted
... show
{offset} (Y/0 {error!})
.. delete
```

```
< expr>
 ... if
< expr>
 ... =
< expr>
 ... Y var
 ... next
< expr>
 ... 0 #
 ... out
(Y = 0)
 ... next
< expr>
 ... 1 #
 ... next
< expr>
 ... -
< expr>
 ... X var
 ... next
< expr>
 ... 1 #
 ... root
|let {Distance}
  X = (3 + 5)
  |let {Altitude}
    Y = (6 / 2)
    (X +
```

```
    (If (Y = 0)
     then 1
     else (X - 1) )
   ) ] ]
... evaluate
15
... done
PI system stopped
OK
> exit{}.

Goodbye.
%
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 2

Dudley Knox Library
Code 0142
Naval Postgraduate School
Monterey, CA 93943 2

Office of Research Administration
Code 012
Naval Postgraduate School
Monterey, CA 93943 1

Chairman, Code 52
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 40

Associate Professor Bruce J. MacLennan
Code 52ML
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 12

Dr. Robert Grafton
Code 433
Office of Naval Research
800 N. Quincy
Arlington, VA 22217-5000 1

Dr. David Mizell
Office of Naval Research
1030 East Green Street
Pasadena, CA 91106 1

Dr. Stephen Squires
DARPA
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209 1

Professor Jack M. Wozencraft, 62Wz
Department of Electrical and Comp. Engr.
Naval Postgraduate School
Monterey, CA 93943 1

Professor Rudolf Bayer
Institüt für Informatik
Technische Universität
Postfach 202420
D-8000 Munchen 2
West Germany 1

Dr. Robert M. Balzer
USC Information Sciences Inst.
4676 Admiralty Way
Suite 10001
Marina del Rey, CA 90291 1

Mr. Ronald E. Joy
Honeywell, Inc.
Computer Sciences Center
10701 Lyndale Avenue South
Bloomington, MI 55402 1

Mr. Ron Laborde
INMOS
Whitefriars
Lewins Mead
Bristol
Great Britain 1

Mr. Lynwood Sutton
Code 424, Building 600
Naval Ocean Systems Center
San Diego, CA 92152 1

Mr. Jeffrey Dean
Advanced Information and Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040 1

Mr. Jack Fried
Mail Station D01/31T
Grumman Aerospace Corporation
Bethpage, NY 11714 1

Mr. Dennis Hall
New York Videotext
104 Fifth Avenue, Second Floor
New York, NY 10011 1

Professor S. Ceri
Laboratorio di Calcolatori
Departimento di Elettronica
Politecnico di Milano
20133 - Milano
Italy 1

Mr. A. Dain Samples
Computer Science Division - EECS
University of California at Berkeley
Berkeley, CA 94720 1

Antonio Corradi
Dipartimento di Elettronica
Informatica e Sistemistica
Universita Degli Studi di Bologna
Viale Risorgimento, 2

Bologna
Italy 1

Dr. Peter J. Welcher
Mathematics Dept., Stop 9E
U.S. Naval Academy
Annapolis, MD 21402 1

Dr. John Goodenough
Wang Institute
Tyng Road
Tyngsboro, MA 01879 1

Professor Richard N. Taylor
Computer Science Department
University of California at Irvine
Irvine, CA 92717 1

Dr. Mayer Schwartz
Computer Research Laboratory
MS 50-662
Tektronix, Inc.
Post Office Box 500
Beaverton, OR 97077 1

Professor Lori A. Clarke
Computer and Information Sciences Department
LGRES ROOM A305
University of Massachusetts
Amherst, MA 01003 1

Professor Peter Henderson
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794 1

Dr. Mark Moriconi
SRI International
333 Ravenswood Avenue
Manlo Park, CA 95025 1

Professor William Waite
Department of Electrical and Computer Engineering
The University of Colorado
Campus Box 425
Boulder, CO 80309-0425 1

Professor Mary Shaw
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213 1

Dr. Warren Teitelman
Engineering/Software
Sun Microsystems Federal, Inc.
2550 Garcia Avenue

Mountain View, CA 94031